Exhibit U

To

Joint Claim Chart

MICROSOFT® PROGRAMMING SERIES

SECOND EDITION

ALL NEW MATERIAL

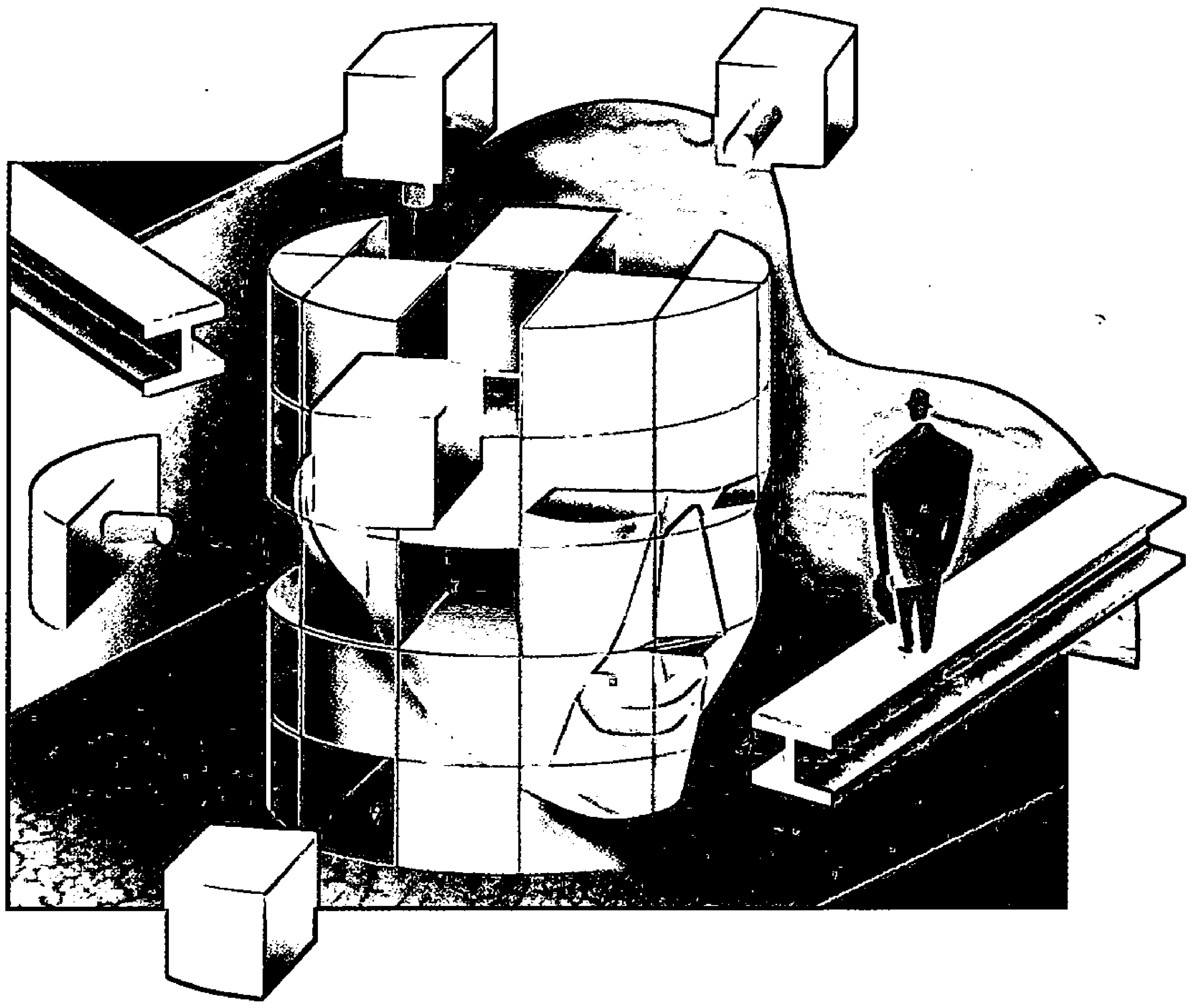
DISC INCLUDED

Includes source code files for over 75 programs

INSIDE OLE

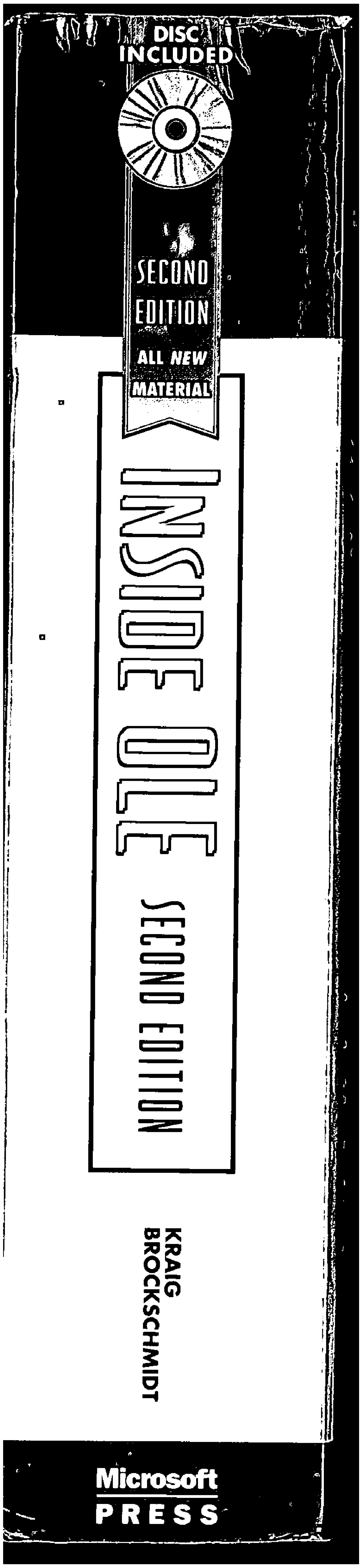
SECOND EDITION

The Inside Track to Building Powerful 32-Bit Component Software with Microsoft's Object Technology



KRAIG BROCKSCHMIDT

Microsoft Press



DISC INCLUDED

SECOND EDITION

ALL *NEW* MATERIAL

INSIDE OLE

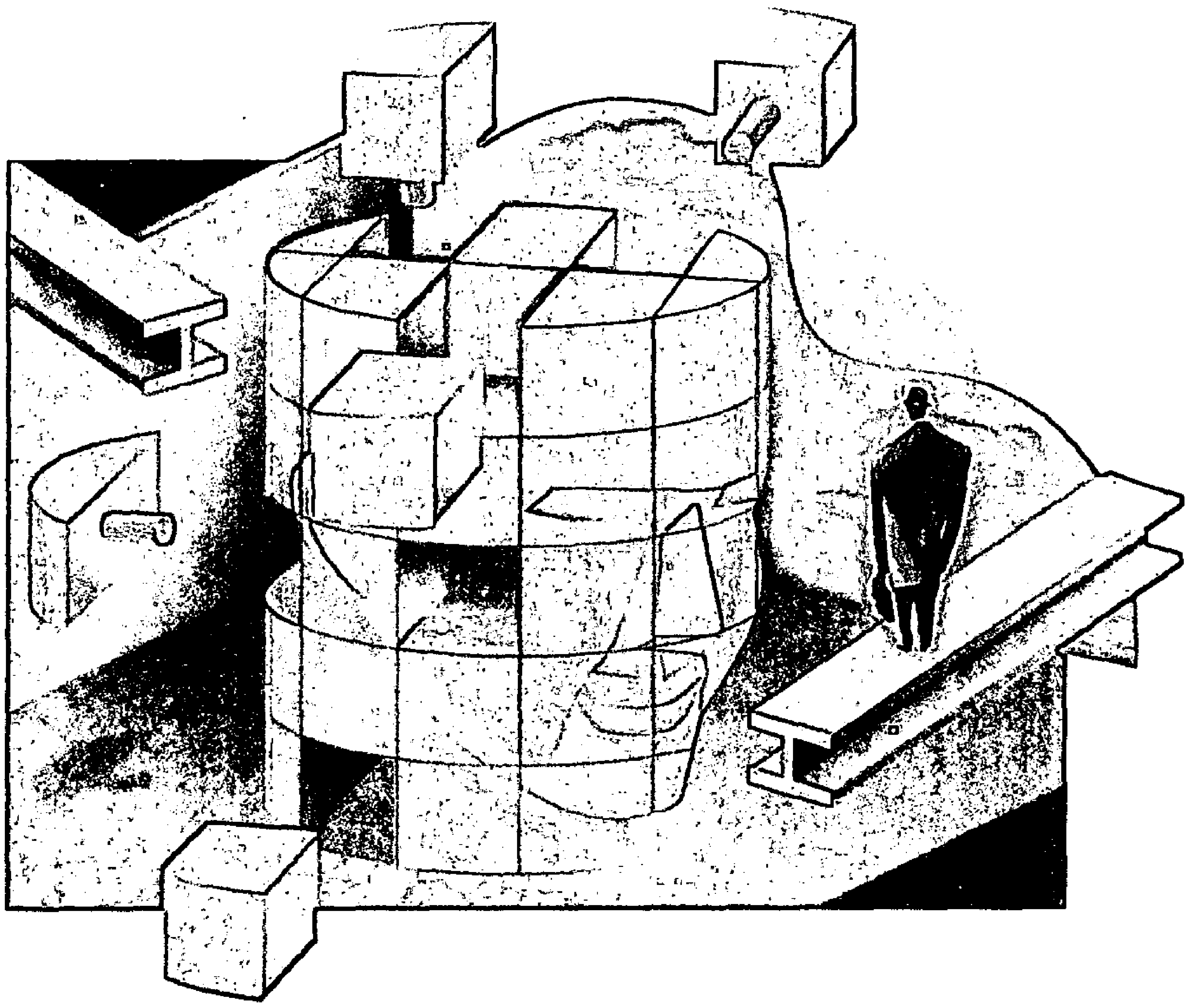
SECOND EDITION

KRAIG BROCKSCHMIDT

Microsoft PRESS

INSIDE OLE

SECOND EDITION



KRAIG BROCKSCHMIDT

Microsoft Press

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Brockschmidt, Kraig, 1968–
Inside OLE / Kraig Brockschmidt. -- 2nd ed.
p. cm.
Includes index.
ISBN 1-55615-843-2
1. Object-oriented programming (Computer science) 2. Microsoft Windows (Computer file) I. Title.
QA76.64.B76 1995
005.75--dc20 95-15969
CIP

Printed and bound in the United States of America.

2 3 4 5 6 7 8 9 QMQM 0 9 8 7 6 5

Distributed to the book trade in Canada by Macmillan of Canada, a division of Canada Publishing Corporation.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office. Or contact Microsoft Press International directly at fax (206) 936-7329.

Acquisitions Editors: Dean Holmes and David Clark
Manuscript Editor: John Pierce
Project Editor: Kathleen Atkins
Technical Editor: Jim Fuchs

demand-loaded code module—such as a DLL or an EXE—that makes a component and its objects available to the outside world. Without a server, the objects remain hidden from external view. The server holds them out on a silver platter and invites clients to partake of them.

The relationship between a client, a server, and the objects that make up a component or a server is illustrated in Figure 1-1. This is similar to the more general definition of any client-server relationship that might use any number of mechanisms to communicate. In the OLE relationship shown here, however, communication happens through OLE objects.

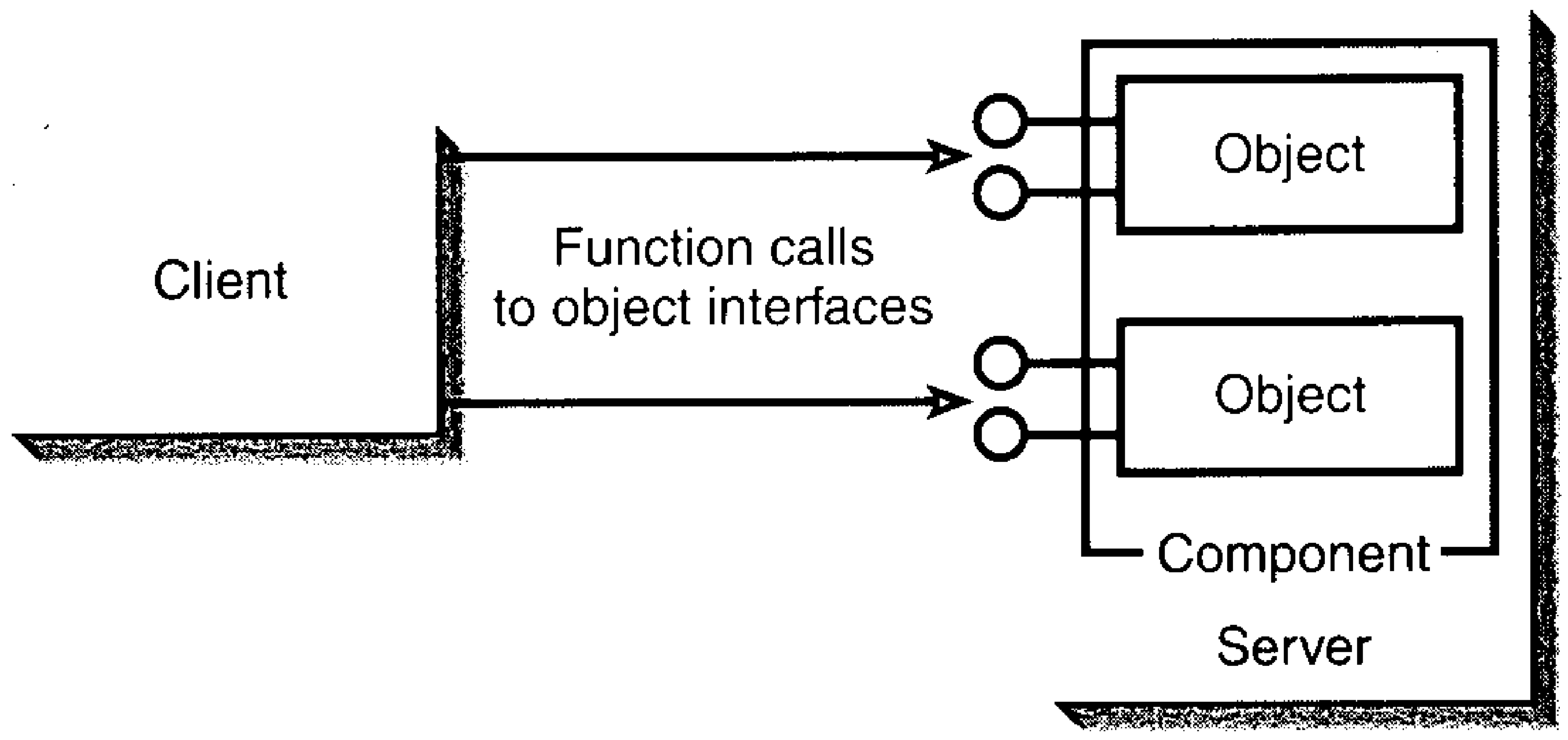


Figure 1-1.
The OLE client-server relationship. A client *uses a component or a service as provided by a* server, *and the communication between client and server happens through OLE objects.*

Object-Based Components: COM

Our definition of OLE describes services as being "object-based." What does this mean exactly? What are "objects"—the communication medium between clients and servers—in the context of OLE? In order to answer this question, we have to look at all the various definitions of the term so we can really understand why an OLE object is what it is. The concepts that form the idea of an OLE object are collectively called the *Component Object Model,* or COM.

Object has to be one of the most bastardized, hackneyed, and confusing terms in the computer industry. Some argue over "real" and "fake" objects. There are academic uses of the term, end-user–oriented uses, and flat out common uses. *Objects* might refer to a methodology, specific techniques in object-oriented programming languages, or icons and other user-interface elements on a computer screen. Then, of course, you have philosophical wackos like me who will argue that the gold-plated Slinky on my desk and the model of the St. Louis Gateway Arch on my bookshelf are also objects.

The Classical Object Definition

When you strip away all the politics, rhetoric, and other baggage and study the original concepts of object-oriented development, you find that an object is an instance of some *class* in which that object is *anything* that supports three fundamental notions:

- *Encapsulation:* all the details about the composition, structure, and internal workings of an object are hidden from clients. The client's view is generically called the object's *interface.* The internals of the object are said to be hidden behind its interface. The interface of the computer keyboard with which I'm typing this book consists of labeled keys, in a specific layout, that I can press to generate a character in my word processor. The internal details of how a keystroke is translated into a character on the screen are encapsulated behind this interface. In the same manner, the internal implementation details of a Smalltalk string object are encapsulated behind the public member functions and variables of that Smalltalk object.
- *Polymorphism:* the ability to view two similar objects through a common interface, thereby eliminating the need to differentiate between the two objects. For example, consider the structure of most writing instruments (pens and pencils). Even though each instrument might have a different ink or lead, a different tip, and a different color, they all share the common interface of how you hold and write with that instrument. All of these objects are polymorphic through that interface—any instrument can be used in the same way as any other instrument that also supports that interface, just as all Slinky toys, regardless of their size and material, act in many ways like any other Slinky. In computer terms, I might have an object that knows how to draw a square and another that knows how to draw a triangle. I can view both of them as having certain features of a "shape" in common, and through that "shape" interface, I can ask either object to draw itself.
- *Inheritance:* a method to express the idea of polymorphism for which the similarities of different classes of objects are described by a common *base class.* The specifics of each object class are defined by a *derived class*—that is, a class derived from the base class. The derived class is said to *inherit* the properties and characteristics of the base class; thus, all classes derived from the same base class are polymorphic through that base class. For example, I might describe a base

class called "Writing Instruments" and make derived classes of "Ballpoint Pen," "Pencil," "Fountain Pen," and so forth. If I wanted to program different "shape" objects, I could define a base class "Shape" and then derive my "Square" and "Triangle" classes from that base class. I achieve polymorphism along with the convenience of being able to centralize all the base class code in one place, within the "Shape" class implementation.

Inheritance is often a sticky point when you come to work with OLE because OLE supports the idea of inheritance only on a conceptual level for the purposes of defining interfaces. In OLE, there is no concept of an object or a class inheriting implementation from another, as there is in C++. But here is the important point: *inheritance is a means to polymorphism and reusability and is not an end in itself.* To implement polymorphic objects in C++, you use inheritance. To create reusable code in C++, you centralize common code in a base class and reuse it through derived classes. But inheritance is not the only means to these two ends! Recognizing this enables us to explore means of polymorphism and reusability that work on the level of binary components, which is OLE's realm, rather than on the level of source code modules, which is the realm of C++ and other object-oriented programming languages.

OLE Objects and OLE Interfaces

As we will explore in this book, objects as expressed in OLE most definitely support the notions of encapsulation, polymorphism, and reusability; again, inheritance is really just a means to the latter two. OLE objects are just as powerful as any other type of object expressed in any programming language, and might be more so. While inheritance and programming languages are excellent ways to achieve polymorphism and reusability of objects or components within a large monolithic application, *OLE is about integration between binary components, and it is therefore targeted at a different set of problems.* OLE is designed to be independent of programming languages, hardware architectures, and other implementation techniques. So there really is no comparison with, and no basis for pitting OLE against, object-oriented programming languages and methodologies; in fact, such languages and methods are very helpful and complementary to OLE in solving customer problems.

The nature of an OLE object, then, is not expressed in how it is implemented internally but rather in how it is exposed externally. As a basis for illustrating the exact structures involved, let's assume we have some software object, written in whatever language (code) that has some properties (data or content) and some methods (functionality), as illustrated in Figure 1-2. Access to these properties and methods within the object and its surrounding server code is determined by the programming language in use.

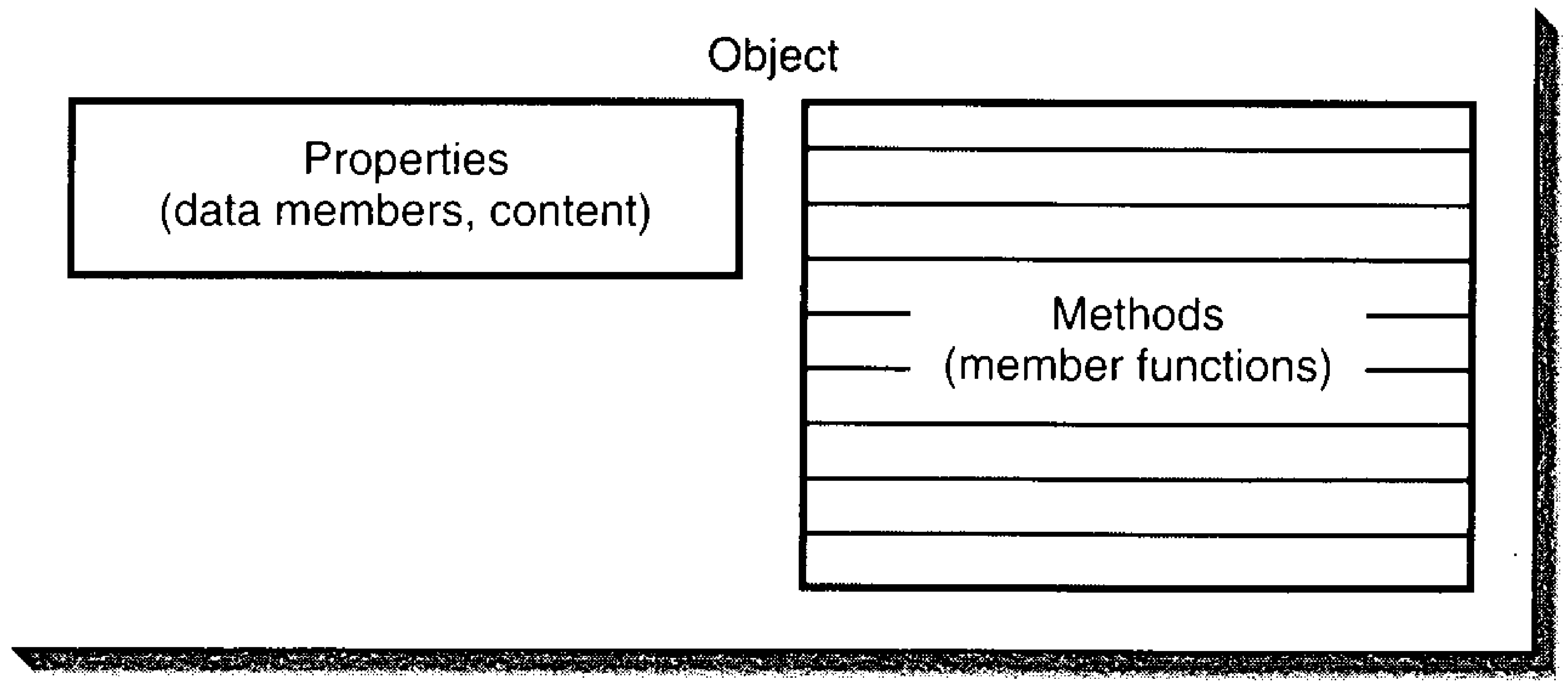


Figure 1-2.
Any object can be seen as a set of properties (data members or content) and methods (member functions).

The accessibility of the object's members in its native programming language is not of interest to OLE. What does concern OLE is how to share the object's capabilities with the outside world, which does not need to correspond at all to the object's internal structure. The external appearance of the object, that is, how clients of this object will access its functionality and content, is what OLE helps you define and implement.

In OLE, you factor an object's features into *one or more groups of semantically related functions,* where each group is called an *interface.* Again, an object can provide multiple interfaces to its client, and this capability is one of OLE's key innovations. Microsoft has already defined many interfaces representing many common features, and many of these interfaces will likely never require revision. For example, the group of functions that describes structured data exchange is easily specified and can thus be a "standard" OLE-defined interface. You are also free to define "custom" interfaces for your own needs without every having to ask Microsoft to approve the design—custom interfaces fit into the OLE architecture the way any standard interface does. We'll explore both standard and custom interfaces throughout this book.

Regardless of who defines an object's interfaces, all access to an object happens through member functions of those interfaces, meaning that OLE doesn't allow direct access to an object's internal variables. The primary reason is that OLE works on a binary component level where you would require complex protocols to control access. (In contrast, programming languages handle such control easily through source code constructs.) In addition, accessing object variables directly usually involves pointer manipulation in the client's address space, which makes it very difficult to make such access transparent across process or machine boundaries unless you stipulate language structures and compiler code generation. On the other hand, because a client gives control to the object through a function call, it is quite easy to transparently intercept that call with a "proxy" object and have it forward the call to another process or machine where the real object is running, and do so in a way independent of languages and compilers.

In the binary standard for an interface, the object provides the implementation of each member function in the interface and creates an array of pointers to those functions, called the *vtable*.[3] This vtable is shared among all instances of the object class, so to differentiate each instance, the object code allocates according to the object's internal implementation a second structure that contains its private data. The specifications for an OLE interface stipulate that the first 4 bytes in this data structure must be a 32-bit pointer to the vtable, after which comes whatever data the object wants (depending on the programming language). An *interface pointer* is a pointer to the top of this instance structure: thus a pointer to the pointer to the vtable. It is through this interface pointer that a client accesses the object's implementation of the interface—that is, calls the interface member functions but cannot access the object's private data. This interface structure is depicted in Figure 1-3.

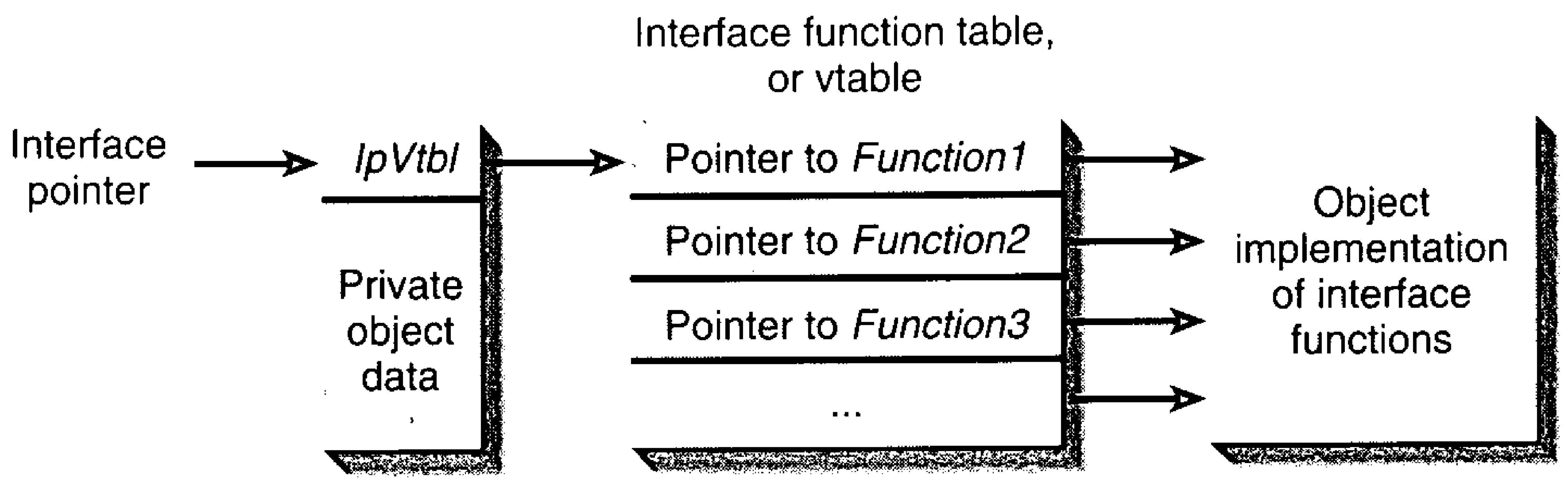


Figure 1-3.
The binary standard for an OLE interface means that an object creates a vtable that contains pointers to the implementations of the interface member functions and allocates a structure in which the first 32 bits are a pointer to that vtable. The client's pointer to the interface is a pointer to the pointer to the vtable.

3. For "virtual function table" because the design of an OLE interface is modeled after the structure of C++ objects that have virtual functions.

If you are familiar with the internals of C++, you'll recognize this structure as exactly that which many C++ compilers typically generate for a C++ object instance. This is entirely intentional on OLE's part, making it very convenient to write OLE objects using C++. In short, if a C++ object class is derived from an OLE interface definition, you can typecast the object's *this* pointer into an interface pointer. Chapter 2 will describe various techniques for doing so. This interface design, however, is merely convenient for C++ programmers. You can easily generate this same interface structure from straight C code, as we'll also see in Chapter 2, and even from assembly language. Since you can express it in assembly language, and because any other programming language can be reduced to an assembly equivalent, you can create an interface from any other language as long as your tools know about OLE and give you a language device through which you can implement or use an interface.

Again I'll point out that because different compilers and languages will store an object's instance data differently in the instance structure, the client cannot directly access that data through an interface pointer. In fact, the client never has any sort of pointer to the object, only to interfaces, because the notion of an object is so variable, whereas the notion of an interface is a binary standard. Besides, direct access—client code manipulating data based on a memory offset—works only when the client and the object share the same address space. In OLE, this is not always the case, so OLE's definition of an interface pointer *type* restricts the use of a pointer at compile time. The definition of the type depends, of course, on the programming language but always allows you to call functions by name through an interface pointer, such as *pInterface->MemberFunction(...)*, and provides type checking on the function's arguments. This is much more convenient than trying to call functions through an array offset with no type checking.

What is highly inconvenient is having to draw the entire binary structure whenever you want to illustrate an object. By convention, interfaces are drawn as plug-in jacks extending from the object, as illustrated in Figure 1-4. When a client wants to use the object through an interface, it must plug into that interface. The electronics analogy is that for a client to use a jack (interface), it must have a plug that fits (code that knows how to use the members of that interface).

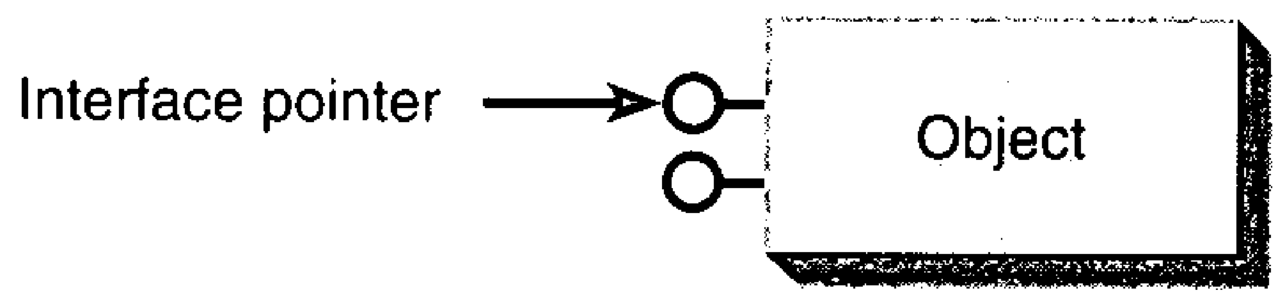


Figure 1-4.
Interfaces are drawn as plug-in jacks extending from the object.

This representation of an object and its interfaces emphasizes an important point: *an interface is not an object.* An interface is merely a channel for the object, and only one of the many channels an object might support. In a "Prolegomenon to Object Metaphysics," we might think of how the philosopher Immanuel Kant, asking how we know something is real, would differentiate objects from interfaces.[4] In such a Kantian analysis, objects are *noumena*—things-in-themselves that are in principle incapable of being known or experienced directly. Interfaces are, on the other hand, *phenomena*—manifestations of objects for sensing experience (your code). Since we can't know objects directly, we tend to refer to them and reify them as their interfaces, but strictly speaking, an object remains conceptual as far as clients of that object are concerned. The client knows only the general "type" of an object as a collection of interfaces that define that type.

As a further reinforcement of this idea, all interfaces are conventionally named with a capital *I* prefix, as in *IUnknown*, *IDataObject*, and so on. The symbolic name of the interface describes the feature of functionality defined in that interface. In addition, you identify an interface at run time not by its textual name but by a binary 128-bit globally unique identifier (globally unique in the real and literal sense). Contrast this to a C++ class, which is only identified at compile time by a text name that is unique only to the compilation.

You should notice that hiding the object behind its interfaces is *exactly* the fundamental notion of encapsulation. OLE also supports the idea of polymorphism between interfaces. All that is needed are two interfaces that share a common subset of functions—a base interface—in their vtables, as shown in Figure 1-5. C++ inheritance works well to define such relationships.

The interface named *Iunknown* has three functions and is the ubiquitous base interface for *every other interface* in OLE. All interfaces are polymorphic with *IUnknown*, as also illustrated in Figure 1-5. This interface represents two fundamental OLE object features. The first is the ability to control an object's lifetime by using reference counting, which happens through the member functions *AddRef* and *Release*. The second feature is navigation between multiple interfaces on an object through a function called *QueryInterface*, as we'll see shortly.

So at least at the interface level, polymorphism is clearly supported in OLE. What is left in the set of fundamental object notions that we've defined here is reusability, and it should be apparent to you that the client of any object can itself be an object. Such a client object can implement any of its

4. My thanks to Mark Ryland (Microsoft) for this philosophical diversion.

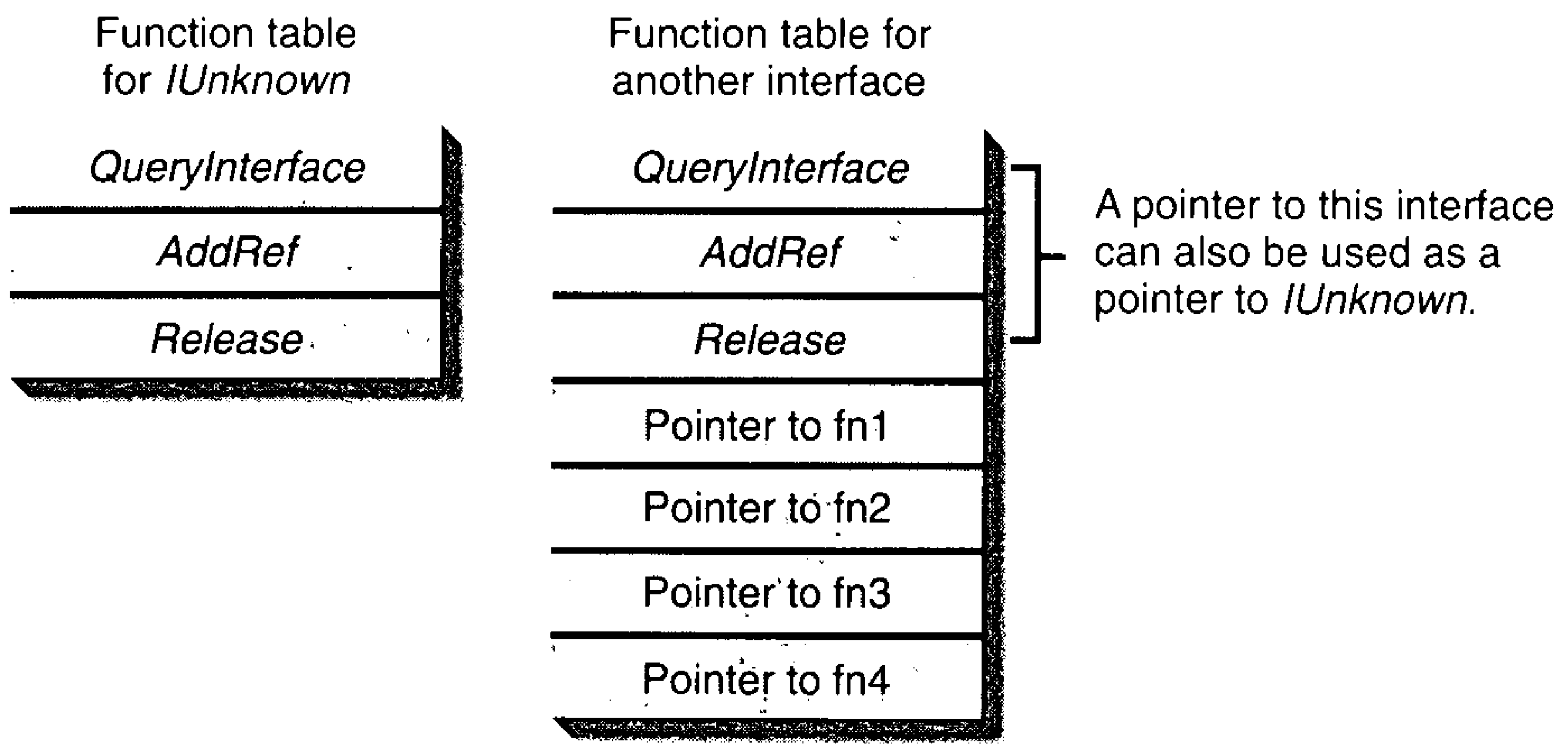

Function table for *IUnknown*
QueryInterface
AddRef
Release
Function table for another interface
QueryInterface
AddRef
Release
Pointer to fn1
Pointer to fn2
Pointer to fn3
Pointer to fn4
A pointer to this interface can also be used as a pointer to *IUnknown*.

Figure 1-5.
Interfaces are polymorphic through a base interface when they both have the same base interface functions at the top of their vtables.

interfaces by using the implementation of another object's interfaces internally, which is reusability by *containment*. The client object contains internally the object being reused, and external clients of the containing object are unaware of such reuse, as it should be. Containment is by far the most common method of reusability in OLE and requires no special support in either object. In some special circumstances, one object might want to directly expose another object's interface pointer as its own, and this requires a special relationship known as *aggregation*. We'll explore reusability through both means in Chapter 2, but for now we can realize that OLE objects support all three fundamental object notions: encapsulation, polymorphism, and reusability. And as we're now ready to discuss, OLE's idea of multiple interfaces is an important and powerful innovation.

Multiple Interfaces and *QueryInterface*

As you read through this book, it is important to remember the difference between an object and an interface: you can implement an object however you want, to do whatever you want; and when you want to share its functionality and content with other components, you can provide as many interfaces as you need, each one representing a subset of the object's features.

Now, even though OLE and C++ share the same binary structure for accessing an object's member functions, the ability to create an object with *multiple interfaces* is extraordinarily powerful. It represents one of OLE's strongest architectural features and is the basis for its notion of *polymorphism between objects* in addition to polymorphism between interfaces, which we have already seen.

Because each interface is a group of semantically related functions, each interface that an object supports represents a specific feature of that object. For example, if an object supports the capability of exchanging formatted data structures, it implements the interface named *IDataObject*, whose member functions describe all the different aspects of exchanging the data, such as set data, get data, enumerate formats, and so on. It is the presence of this interface that describes the object's data exchange capacities.

Herein lies OLE's architectural advantage over single-interface models such as C++: objects can describe their features on a higher level of abstraction than individual member functions, which enables a client to ask the object whether it supports a particular feature before that client attempts to use such a feature. Contrast this to C++ techniques, which might require that a client attempt to call a function simply in order to check whether that function will work. The ability of a client to ask an object about support for a feature decouples the act of testing for functionality from the act of invoking the functionality.

The function *IUnknown::QueryInterface* (using C++ syntax) is the decoupling mechanism that a client uses to navigate through multiple interfaces. Because *QueryInterface* belongs to *IUnknown*, and because all interfaces in OLE are derived from and are polymorphic with *IUnknown*, *QueryInterface* is universally available through whatever interface pointer a client might have. The interfaces available through the same implementation of *QueryInterface* are said to be implemented *alongside* one another. If you see a reference to "interface A is implemented alongside interface B," it means that you can use *QueryInterface* to get from A to B and from B to A.

Whenever a client accesses any object, it can always obtain an initial *IUnknown* pointer for that object. But because the client can only call *IUnknown* functions through this pointer, it can't do a whole lot with the object. In order to use any other object feature—any other interface—the client must first *ask* the object whether it supports that feature by calling *QueryInterface*. In making this call, the client passes the unique identifier a globally unique identifier of the interface it would like to access. If the object supports that interface, it will return a pointer to that interface to the client; otherwise, it returns an error. If support is there, the client is given the exact interface pointer through which it can then call the member functions of that interface and access that feature. If support is not there, the object provides no such pointer, thereby disallowing all calls to unsupported features. An illustration of the *QueryInterface* process is shown in Figure 1-6.

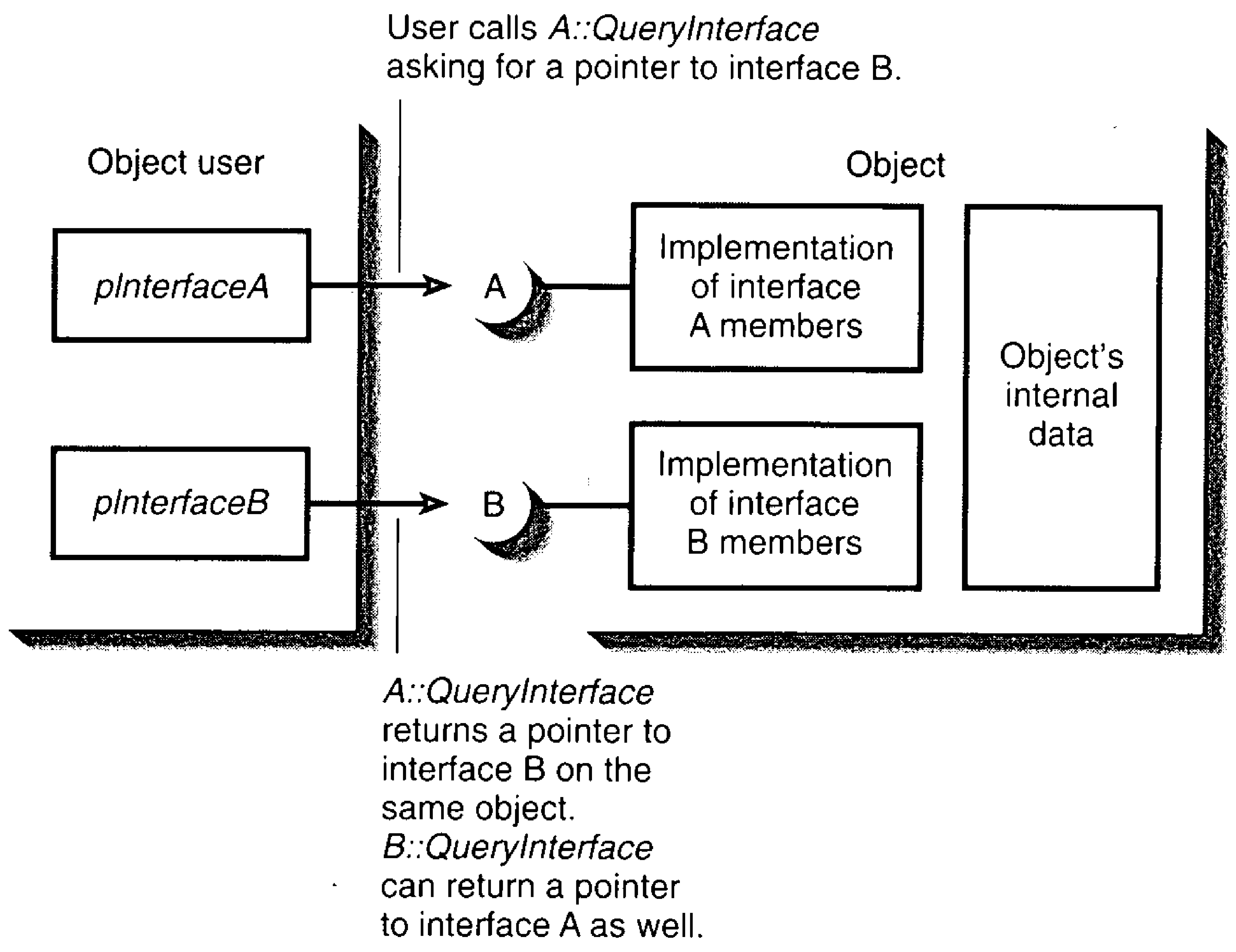

User calls *A::QueryInterface* asking for a pointer to interface B.
Object user
pInterfaceA
pInterfaceB
Object
A
B
Implementation of interface A members
Implementation of interface B members
Object's internal data
A::QueryInterface returns a pointer to interface B on the same object. *B::QueryInterface* can return a pointer to interface A as well.

Figure 1-6.
The QueryInterface *function navigates through all the interfaces on an object.*

So *QueryInterface* is a tight coupling between asking an object whether it supports a feature and the ability to access that feature: *a client must have an appropriate interface pointer to access a feature, and the only way to obtain the pointer is by calling* QueryInterface—*you have to ask!*

Let this sink in for a while, and you might begin to realize the real importance of this simple mechanism. *QueryInterface* and the idea that an object's functionality is factored into *interfaces* rather than single *functions* provide what is called "robust evolution of functionality over time." This is the ability to take a component and its constituent objects, add new objects to the component and new interfaces to its objects, and redeploy the component into a *running* system without breaking compatibility with existing clients. Because new features are added in the form of new uniquely identified interfaces and existing clients will never ask for interfaces that they do not understand, new interfaces do not interfere with any existing interfaces. As far as existing clients are concerned, the objects have not changed, but new clients that do understand those additional interfaces can take full advantage of the new features. In this way, you have components and clients that can evolve independently over time, through many revisions, retaining full compatibility with the past without stunting future improvements.

A key point in all of this is that a change to an object or a component *requires absolutely no recompilation or changes whatsoever to existing clients.* Contrast this with a change to a C++ base class, which always requires a recompilation of any derived classes. In OLE, you never have to update an object client just because the object changes. Certainly any existing client will not use the new features of the object, but as soon as that client is independently updated to ask for those new interfaces, it can take advantage of supporting objects immediately without requiring any changes to the objects themselves.

We'll see a concrete example of this sort of independent evolution of an object and a client in Chapter 2. What is still left to mention here is the idea of polymorphism between objects. Because an interface is defined as a fixed group of member functions, an interface implemented on any object has exactly the same functions and semantics as the same interface implemented on any other object. In other words, a client can call those member functions without having to know exactly what type of object it's really talking to. This is polymorphism: any two objects are polymorphic through the interfaces they have in common. The client can always call each member function in the interface, but what the object does in response to the call can vary within the design of the interface function. For example, with the function *IViewObject2::Draw,* a client can ask an object to draw its visual presentation to the screen or a printer. What the object will actually draw depends on the object and its state data, but the client's *intent* to draw the object is constant.

OLE takes advantage of this polymorphism in a number of its higher-level technologies, such as those dealing with compound documents and custom controls. Every OLE control, for example, supports the same set of interfaces that define a control, and so a *control container*—a client that specifically works with controls—treats all controls polymorphically through those interfaces. The container need not care about the specific types of controls: they're all just controls.

The notion of multiple interfaces for OLE objects has tremendous power and implications. To my knowledge, this extraordinary facility is not part of any other object model, programming language, or operating system technology. The idea that you can factor an object's capabilities into distinct, feature-oriented groups opens all sorts of opportunities to create innovative components and offers a promising future of interoperability and integration between components that has never before been realized.

OLE Technologies, Features, and Services

To reiterate our definition, OLE is a unified environment of object-based services with the capability of both customizing those services and arbitrarily extending the architecture through custom services, with the overall purpose of enabling rich integration between components. Since we now understand how components express their capabilities through interfaces, we can take a look at the individual technologies within the rest of OLE.

OLE as a whole (including recent and future enhancements) is made up of a number of Win32-style API functions (those with global names that are direct calls into system DLLs) as well as a large number of interfaces. Each interface expresses a certain feature through some number of member functions. OLE currently has no fewer than 120 API functions and no fewer than 80 interfaces that average 5 to 6 member functions each. A little quick math, and you end up with almost 600 individual functions. That's a tremendous amount of functionality! But all of it falls into three categories:

- Access to OLE-implemented components (which include helper functions to make programming tasks easier)
- Customization of OLE-implemented components
- The capability of extending the environment through custom components with a wide range of possible features

In other words, OLE implements a fair amount of functionality itself and, in many cases, allows direct customization of that functionality through implementations of small isolated objects that are plugged into those components. Where OLE doesn't provide a component—most of its components are very general—OLE supports the creation of custom components, making them available to all clients through the same architecture. Many of OLE's interfaces are usually implemented on the objects in these custom components, and many of OLE's API functions assist in the operations that are involved within those interfaces.

There has been some confusion in OLE's past about who or what implements this or that interface, but this is really a meaningless question. The key question is who or what implements a particular component, and once you've answered that question, you can then ask what features that component supports and what interfaces are used to supply those features. It is true that some of OLE's components exclusively control certain types of objects and thus are the only implementers of certain interfaces. This has led some people to think that only one type of object ever implements a specific interface, but the truth is that most interfaces can be implemented by any object in any component whatsoever.

So to better understand this difference between components, the objects involved, and the interfaces on those objects, let's look at all of the various technologies within OLE, discussions of which comprise the contents of the rest of this book:

Type Information (Chapter 3) A means to completely describe an object along with all of its interfaces, down to the names and types (including user-defined types) of each argument to each named function in each interface.

Connectable Objects (Chapter 4) The ability to create outgoing interfaces for an object, such as notifications and event sets.

Custom Components and COM (Chapter 5) The ability to create an optionally licensed component that extends the available services to clients.

Local/Remote Transparency (Chapter 6) The ability to transparently integrate clients and components across process and machine boundaries as if they were all in the same process.

Structured Storage and Compound Files (Chapter 7) A powerful and shareable means to deal with permanent storage that offers benefits of incremental access and transactioning.

Persistent Objects (Chapter 8) The interfaces and protocols necessary to share storage between components and clients.

Naming and Binding: Monikers (Chapter 9) The encapsulation of a name of an object or a process with the intelligence necessary to work with that name.

Uniform Data Transfer (Chapter 10) The ability to exchange data structures between components and receive notifications about data changes.

Viewable Objects and the Data Cache (Chapter 11) The ability to have an object control its visual representation on any output device and the ability of a client to cache those representations for use when the object is unavailable.

OLE Clipboard (Chapter 12) Support for the familiar operations of Cut, Copy, and Paste using the mechanisms of Uniform Data Transfer.

OLE Drag and Drop (Chapter 13) A mouse-oriented means of performing the same operations as the clipboard.

OLE Automation (Chapters 14 and 15) The ability to expose an object's methods and properties as individual entities in a late-bound manner, also enabling cross-application macro programming.

Property Pages, Changes, and Persistence (Chapter 16) A user interface for manipulating properties, a mechanism for notifying clients about property changes, and standards for the serialization of a set of properties into persistent storage.

OLE Documents: Embedding and Linking (Chapters 17–21) The basic protocols for the creation and management of compound documents, by which active content objects are manipulated in windows separate from the document itself.

OLE Documents: In-Place Activation (Chapters 22 and 23) An extension to embedding in which the active object is manipulated in place within the compound document. This is also called *visual editing* or *in situ editing*.

OLE Controls (Chapter 24) The ability to create custom controls as OLE objects and the protocols for managing controls as in-place active objects within a document or form. OLE Controls includes events, property pages, and keyboard mnemonics.

Futures (Chapter 25) A look ahead to future enhancements and additions to OLE and what they will mean to component software.

Each OLE technology is described briefly in the following sections, and the rest of the chapters in this book deal with each of these technologies in detail.

Keep in mind that absolutely all of these technologies are built on the idea of components and objects and interfaces called the Component Object Model, or COM. Each technology has specific interfaces that apply to it, and some of the higher-level protocols such as OLE Documents and OLE Controls even involve *groupings* of the interfaces from other technologies. Because of these relationships, the technologies in this list build on one another, as illustrated in Figure 1-7 on the following page.

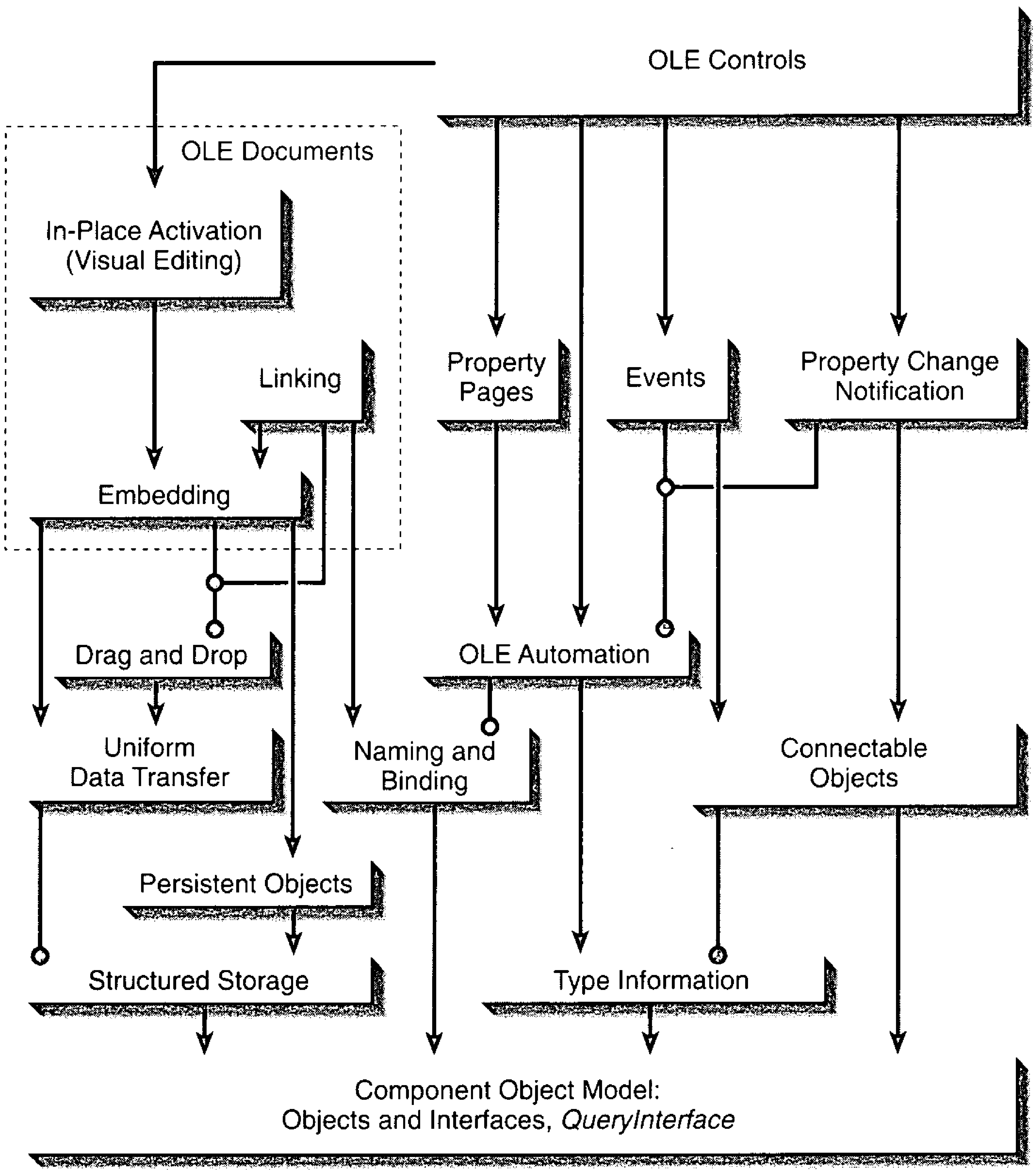


Figure 1-7.
OLE technologies build on one another, with COM as the foundation. An arrow indicates dependency; a circle indicates a possible use but not a requirement.

You can see that COM, as well as custom components and local/remote transparency, which are generally considered part of COM, form the underlying basis for everything else in OLE. The lower a technology appears on this chart, the more generic or general purpose it is and the less visible it is to an end user. The higher technologies are considered more specific; they are generally more complex, and they usually involve more user interface the higher you go. In other words, the highest technologies are the most visible to the end user, but when you set out to incorporate these technologies into a piece of software, it makes the most sense to work from the bottom up because in doing so you'll build a foundation of code that is readily usable when you work on the higher-level technologies. I most definitely encourage this sort of approach to learning OLE, as the organization of this book reflects.

Is OLE the Only Way?

Undoubtedly you will come across situations in your development efforts in which you have a problem that perhaps some of these OLE technologies can solve. The question is then whether OLE is the *only* solution to that problem. In all likelihood, there are many possible methods that you might use to solve the problem—OLE is in no way required as the solution *unless* you are dealing with an integration problem among components from multiple vendors. In that case, you want to adhere to the standards and interfaces that make up the various OLE technologies. In other words, integration among arbitrary components that were not known to each other during development requires standards, and that is what OLE provides. If your problem involves integration among only those components that you write yourself, you can design whatever solution fits your needs. Remember, however, that everything in OLE has already been through a rigorous design and open review cycle. Using OLE in even a closed system makes sense because you do not have to struggle with the same problems that OLE solves already.

Type Information

In the earlier material about *QueryInterface*, you might have thought of some very common questions. How can a client obtain just a list of all of an object's interfaces short of calling *QueryInterface* for every known interface under the sun? How could an object browser accomplish this without even instantiating objects in the first place? How can a client learn about the names of interface member functions as well as the names and types of the arguments to those functions? One answer to this last question is a header file that contains the function signatures for the interface in question. Header files, however, contain compile-time information. How can you obtain the same information at run time?

The answer to these questions is known as *Type Information*—literally, a collection of information about data types, interfaces, member functions (return types and arguments), object classes, and even DLL modules. Type information is really just a complex set of nested data structures: one structure describes an object and its interfaces, the interfaces have data structures describing the individual member functions, the functions have structures to describe the individual arguments, and these in turn have structures to describe their various attributes and data types. In reality, this is as much, if not more, information than you could hope to glean from any header file.

To deal with this complexity, OLE provides native services to navigate through type information as well as to create it in the first place. In other words, all creation and manipulation of type information happens through OLE-implemented objects and their interfaces, so you rarely have to deal with these complex data structures directly. These OLE components are the topic of Chapter 3.

All type information is stored in an entity called a *type library,* which can contain information for any number of objects, interfaces, data types, and modules. Creating a type library happens through an object that implements *ICreateTypeLib* and creating elements in the library happens through an object with *ICreateTypeInfo.* The implementation of these objects is provided as a standard OLE service, so you will never implement these yourself. In fact, you will generally never have occasion to even use them as a client because the OLE SDK includes a tool called MKTYPLIB.EXE (Make Type Library), which takes a text file you write in an Object Description Language (ODL), parses it, and makes all the appropriate calls to *ICreateTypeLib* and *ICreateTypeInfo,* generating a file that contains all the binary data structures for your type library.

Accessing the information contained in a type library is accomplished through two similar OLE-provided objects. The library exposes *ITypeLib,* and the elements in it expose *ITypeInfo.* OLE offers a number of means through which you can load a type library into memory and obtain the *ITypeLib* pointer for it, as we'll see in Chapter 3. After you have an *ITypeLib* pointer, you can easily obtain the *ITypeInfo* for an object class (given a unique class identifier for the object), and through this interface you can obtain a list of the interfaces the object supports. All of this can be done without ever instantiating an object, but it does require a little information in the system registry to associate an object class with a type library.

If a client wants to retrieve the same information from an object that is already running, it can query for an interface named *IProvideClassInfo*. The presence of this interface means that the object can directly provide the client with the *ITypeInfo* that describes the object as a whole, without the bother of having to go out and find a type library.

I should point out that at the time of writing, type information is available for only a few objects outside of OLE Automation or OLE Controls (where it is required). However, type information is universal—it can describe any object and any interface, so as time passes I expect to see type information become available for most objects. Any type-related information about anything in OLE can be stored in a type library, so such a library is *the* repository for information about objects and interfaces.

Connectable Objects

The set of interfaces that a client can access through *QueryInterface* forms an object's *incoming* interfaces when calls to those interfaces come into the object. However, this alone is not entirely sufficient to completely describe the possible features of objects: sometimes an object will want to notify external clients when events happen in the object—for example, when data changes or when a user performs an action such as clicking the mouse on the object. In order to send such notifications (or to "fire events," as it is sometimes called), the object supports *outgoing* (sometimes called *source*) interfaces, illustrated as an arrow coming out of an object, as shown in Figure 1-8.

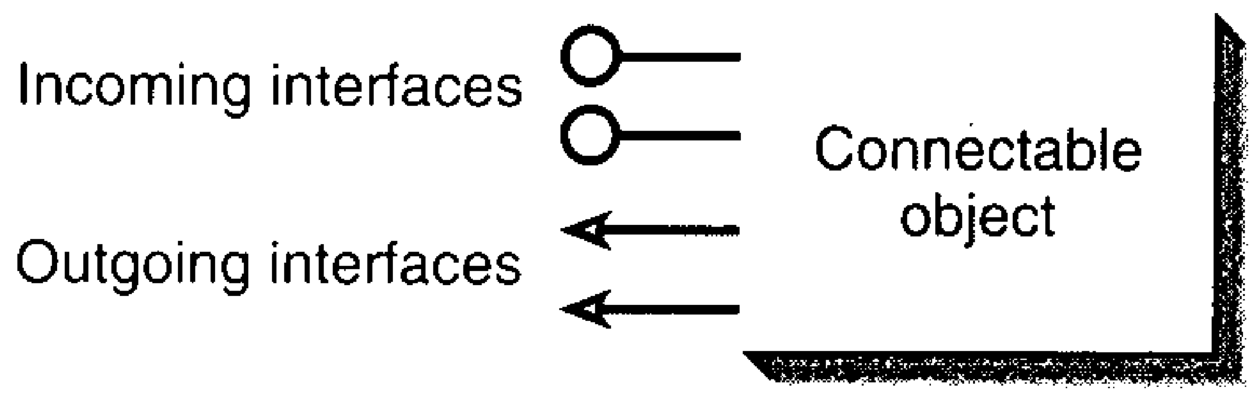


Figure 1-8.
An object's outgoing interfaces are drawn as outgoing arrows alongside the jacks that represent incoming interfaces.

Outgoing interfaces are not ones that the object implements itself. Instead, the object is a client of the interface as implemented on an external object. The most frequent use of this sort of relationship is the forming of a two-way channel of communication between two components, where each is both client and object. In order not to confuse the terminology, the object that sends the notifications is called the *source,* and the client that receives the notifications is called the *sink,* as illustrated in Figure 1-9.

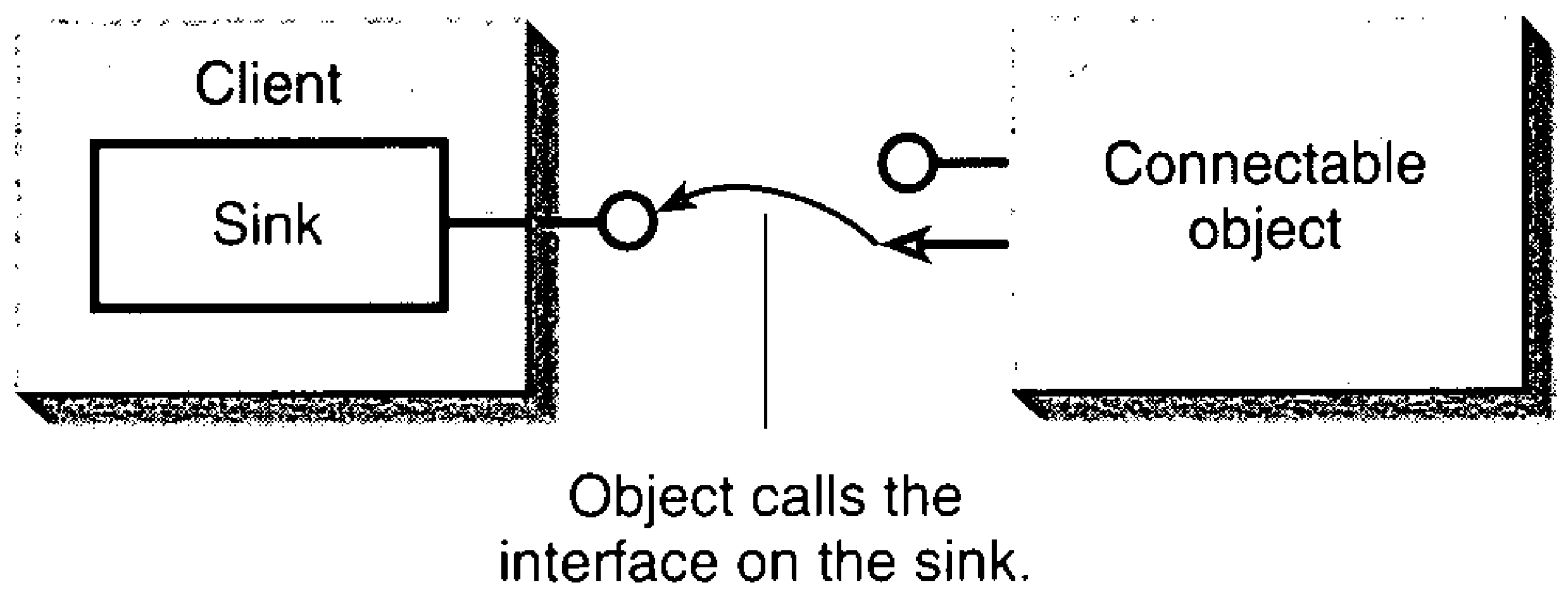


Figure 1-9.
A client that implements an object's outgoing interface is a sink for the object's notifications and events.

What is special in this outgoing interface relationship that differentiates it from the usual incoming interfaces is that the object generally defines the exact outgoing interface itself. This is why we still call it the *object's* interface even though the client implements it—the object defines outgoing interfaces as part of its feature set.

The exact mechanism used to establish this two-way channel of communication is the technology called Connectable Objects, which is the topic of Chapter 4. When any object wants to express the fact that it has outgoing interfaces, it implements an interface named *IConnectionPointContainer,* and in doing so it qualifies as a connectable object. When a client wants to check whether an object has outgoing interfaces, that client will call *QueryInterface,* asking for *IConnectionPointContainer.* With this interface, the client can browse through individual entities called *connection points;* each connection point represents a single outgoing interface. These connection points are small objects themselves (with their own *IUnknown* behavior), managed within the larger object, each of which implements an interface named *IConnectionPoint.* It is through this interface that the client can ask the connection point for the identifier of its outgoing interface and give the connection point the interface pointer through which the connectable object sends notifications (that is, calls interface member functions). In other words, to establish the two-way channel, the client has to be able to pass its own interface pointers to the connectable object, and connection points are the means for doing so.

Some outgoing interfaces involved in a two-way dialogue are known to both sides at compile time. These are known as standard event sets, or standard notification interfaces, which are defined in header files. In other cases, however, the client does not know the exact details of the interface until run time, which requires the object to supply type information.

The idea of events, which connectable objects provide, is a very powerful one. So many things—controls, menus, the keyboard, the mouse, modems, power failures, bothersome relatives—generate events. Anything that somehow has something to say is an event source, and connection points are how you listen. What you do when you hear that a certain event has occurred is completely open. Events are really the triggers that make things happen and that transform user actions (such as typing, moving the mouse, and speaking into a microphone) into information that a piece of software can use as a stimulus to drive a particular response. Stimulus-response makes things seem *alive.* This makes events a key to dynamic software, and connectable objects are how events are implemented in OLE.

Custom Components and COM

The Component Object Model (COM) is a specification (hence "model") that describes, besides the notions of objects, interfaces, and *QueryInterface,* the mechanisms necessary to access a custom component given only a unique identifier.[5] This identifier, called a class identifier (or CLSID), identifies the top-level object within the component, so what COM is really providing is a mechanism to instantiate an object of a given class. This sounds a lot like the C++ *new* operator, but in OLE the client calls a COM API function *CoCreateInstance* with the CLSID to obtain an interface pointer to that object rather than calling *new* with a C++ class name, which results in an object pointer. Of course, we're talking about two different species of objects here, but the analogy still holds.

Not all objects in OLE need to support this sort of creation process, but when support is there, it is the responsibility of the object or component *server* to package the object appropriately, which is the primary topic of Chapter 5. The mechanics of this depend on whether the server is a DLL or an EXE, but a client always starts the process by calling *CoCreateInstance* with a CLSID. COM uses information stored with the CLSID in the system registry

5. In dealing with specific COM-related features, it is conventional to use the term *COM* in place of or interchangeably with the term *OLE* because COM identifies the core part of OLE that provides for object creation and local/remote transparency. COM API functions are all named with a *Co* prefix.

to locate the server module. It then gets that server into memory (loads the DLL or launches the EXE) and asks the server to create the object, returning an interface pointer. The act of creation involves an object known as a *class factory*, which implements the interface *IClassFactory*, whose purpose in life is to manufacture objects of a particular CLSID. This object is part of the server, not part of the component that the client is trying to access, although a client can access the class factory directly if it wants.

In any case, the client is completely isolated from how the object is structured inside a server and from all considerations about the boundaries between it and the object created. The object could be *in-proc* (in-process, from a DLL server), *local* (out-of-process, from an EXE server), or even *remote* (from a DLL or an EXE that is running on another machine).[6] In any case, COM's local/remote transparency (see the next section) means that the client doesn't have to care where the object is running—it simply accesses its features through interface pointers, and those calls are transparently routed to other processes or machines as necessary. Clients do, however, have control over what sort of boundaries are involved; for example, a client can specify only in-process or only out-process.

So far I have made no mention about what interfaces the various objects in a component might support, and this is intentional: the custom component facilities in COM make no restrictions on what interfaces an object might have (although local/remote transparency does at times). These facilities can be used anytime and anywhere a client might need to access any sort of component through a CLSID.

As part of these same mechanisms, COM also provides for what is called *emulation* of one component class by a component from a different class. This is accomplished by mapping one CLSID to another in the system registry. What this does is allow one component to be installed over another and completely replace it without requiring changes to any existing clients that might have hard-coded or persistently stored CLSIDs. When the client instantiates a component of the original CLSID, COM automatically uses the other component that is emulating the first. This is useful for providing alternative implementations of the same component and is a key factor for installing new versions of a component without breaking its existing clients.

6. Note that at the time of writing, OLE/COM does not yet support remote objects, also called *distributed services*, as a shipping technology because there are still some things to be worked out.

Part of the enhancements to OLE that were shipped with the set of technologies called OLE Controls is the ability to license the creation of components. This is handled through an alternative class factory interface named *IClassFactory2*, which provides a license key that is necessary to create objects later. This class factory would provide such a key only when running on a machine with a fully licensed installation. If the server is illegally copied to another unlicensed machine, the server will refuse outright to create anything.

Local/Remote Transparency

I mentioned that OLE, or more accurately COM, allows components to be implemented in either DLL or EXE servers, which means that those components can run either in the same process as the client or in a separate process. When Microsoft enables distributed services and remote objects, components will then be able to run in another process on another machine. In all cases, COM's architecture enables a client to communicate with any object through an interface pointer in the client's address space. When the object is in-process, calls go directly to the object's implementation. When there is a process or a machine boundary between the client and the object, the call cannot be direct because the client's pointer is meaningless outside its process. How does COM route calls from the client to the object's real implementation, wherever it is running, and do so transparently to the client?

COM's Local/Remote Transparency is the technology that makes calls to a *local* or a *remote* object identical to a call to an *in-process* object as far as the client is concerned.[7] The way it works, shown in Figure 1-10 on the next page, is that the client always has a pointer to some in-process implementation of the interface in question. If the object is truly in-process, that implementation is the object's. If the object is local or remote, the implementation is part of an in-process object *proxy*. When the client makes a call to the interface, the proxy takes all the arguments to that function and packages them in some portable 32-bit data structure (which involves copying data structures, strings, and so forth) and generates some sort of remote procedure call (RPC)[8] to the other process (or machine). In that other process, a *stub*, which maintains the real interface pointers to that object, receives the call, unpacks the data structure, pushes arguments on the stack, and makes the call to the object. When

7. Even as COM doesn't support the remote case at the time of writing, it was intentionally designed to handle remote objects through the same architecture.

8. Under Windows 3.1 (16 bits), the mechanism is a private implementation called LRPC, for "lightweight remote procedure calls," which is strictly local to a machine. On 32-bit platforms, OLE uses the true underlying system RPC as defined by the Open Software Foundation (OSF) Distributed Computing Environment (DCE).

such as S_FALSE (which is 1), the comparison will be wrong. There are only a few cases in which you *really* want to know whether the function returns *exactly* NOERROR. Otherwise, you should always use SUCCEEDED and FAILED.

You may have occasion to display a useful message to an end user when you encounter an error represented by an HRESULT. For this purpose, OLE offers the function *FormatMessage* that returns a user-readable message for any given HRESULT, localized to the user's language as appropriate. See the *OLE Programmer's Reference* for more information on this function.

Interface Attributes

The first and foremost concept surrounding an interface is that it is a form of contract between the client using the interface and the object implementing it. This contract means that when a client has a pointer to an interface, the client can successfully call every member function in that interface. In other words, when an object implements an interface, it must implement every member function to at least return *E_NOTIMPL*. This means that after a client has obtained a pointer to an interface from a call to *QueryInterface*, it no longer has to ask the object whether the functions in that interface are callable—they are. The functions may not actually *do* anything, but they can be called. That is the nature of the contract.

Given that, there are four other important points about interfaces:

- Interfaces are not classes. An interface is an abstract base class, so it is not instantiable. It is merely a template for the correct vtable structure for that interface, providing names and function signatures for each entry in the vtable—an interface definition carries no implementation. It must be implemented in order to be usable. Furthermore, different object classes might implement an interface differently yet be used interchangeably in binary form, as long as the behavior conforms to the interface specification (such as two objects that implement *IStack*, where one uses an array and the other a linked list).

- Interfaces are not objects. Interfaces are the means to communicate with objects, which are otherwise intangible entities. The object can be implemented in any language with any internal structure so long as it can provide pointers to interfaces according to the binary structure. Because all interfaces work through function calls, the object can expose its internal state only through such functions.

- Interfaces are strongly typed. At compile time, the compiler will enforce unique names for interfaces that identify variable types. At run time, an interface is globally unique by virtue of its IID, thereby eliminating the possibility of collisions with human-readable names. Interface designers must consciously assign an IID to any new interface, and objects and clients must consciously incorporate this IID into their own compilations to use that interface at run time. In this way, collisions cannot happen by accident, and this leads to improved robustness.
- Interfaces are immutable. Interfaces are never versioned; revising an interface by adding or removing functions, changing argument types, or changing semantics effectively creates a new interface because it inherently changes the contract of the existing interface. Therefore, the revision must be assigned a new IID, making it as different from the original interface as any other. This avoids conflicts with the older interface. Objects can, of course, support multiple interfaces simultaneously so that the objects have a single internal implementation of the capabilities exposed through two or more similar revisions of an interface while still fulfilling their contractual obligations to all clients. This approach of creating immutable interfaces and allowing multiple interfaces per object avoids versioning problems.

Just because a class supports one interface, there is no requirement that it support any other. Interfaces are meant to be small contracts that are independent of one another. There are no contractual units smaller than interfaces. However, specifications or protocols such as OLE Documents and OLE Controls are *higher* contractual units than interfaces; objects must implement a related set of interfaces as defined by the specification of a certain prototype. See "Class, Type, and Prototype" on page 64. It is true that all compound document objects or OLE controls will always implement the same basic set of interfaces, but those interfaces themselves do not depend on the presence of the other interfaces. It is instead the clients of those objects that depend on the presence of all the interfaces.

The encapsulation of functionality in objects accessed through interfaces makes COM/OLE an open, extensible system. It is open in the sense that anyone can provide an implementation of a defined interface and anyone can develop a client that uses such interfaces. It is extensible in the sense that new or extended interfaces can be defined without changing existing clients or components, and those clients that understand the new interfaces can exploit them on newer components while continuing to interoperate with older components through the old interfaces. Still better is the fact that no underlying changes to COM or OLE are required to support your own custom interface designs, as we'll see in Chapter 6. Because of that, you can extend the system without ever having to involve Microsoft, a big change from previous service architectures in which any change in features meant a change to the system API. You don't have to wait for Microsoft any longer—you can innovate as fast and as often as you want and control your own destiny!

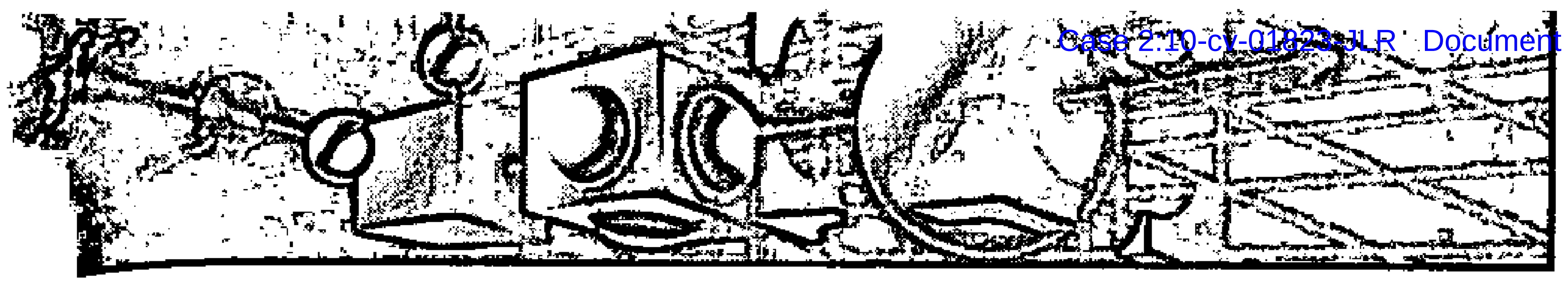
IUnknown: The Root of All Evil

The *IUnknown* interface is the one interface that all objects must implement, regardless of what other interfaces are present. *IUnknown* is what defines *objectness* in OLE. An Object's *IUnknown* pointer value is what gives that object *instance* its run-time identity. Implementing *IUnknown* presents little challenge because *IUnknown* is the base interface for every other interface in OLE. By virtue of implementing any interface, you'll implement *IUnknown* to boot. In some cases, as we'll see in "Object Polymorphism and Reusability" later in this chapter, you'll need to implement two different sets of *IUnknown* member functions, but most of the time you'll implement only one set.

This interface itself encapsulates two operations: the control of an object's lifetime (or life cycle, as it is sometimes called, which sounds like a piece of horrendous exercise equipment, so I prefer the first term) and the navigation of multiple interfaces:

IUnknown Member Function	Result
ULONG AddRef(void)	Increments the object's reference count, returning the new count.
ULONG Release(void)	Decrements the object's reference count, returning the new count. If the new count is 0, the object is allowed to free (delete, destroy) itself, and the caller must then assume that all interface pointers to the object are invalid.
*HRESULT QueryInterface (REFIID riid, void **ppv)*	Asks the object whether it supports the interface identified by *riid* (an IID reference); a return value of NOERROR indicates support exists, and the necessary interface pointer is stored in the out-parameter **ppv*. On error, E_NOINTERFACE says the object does not support the interface.

The following sections examine reference counting and *QueryInterface* in more detail.



CHAPTER FOUR

Connectable Objects

It is the province of knowledge to speak, and it is the privilege of wisdom to listen.
—Dr. Oliver Wendell Holmes

In Chapter 2, we explored the notion of incoming interfaces for an object and the *QueryInterface* function that manages these interfaces. "Incoming," in the context of a client-object relationship, implies that the object "listens" to what the client has to say. In other words, incoming interfaces and their member functions are like an object's sensory organs—its eyes, ears, nose, and nerve endings—which receive input from the outside. But there is only so much you can say in a one-sided conversation.

Objects are fairly tolerant of loquacious clients, so they usually don't mind listening. Many objects, however, have useful things to say themselves, and this requires a two-way dialogue between object and client. Such two-way communication involves *outgoing* interfaces—the different languages that an object can speak through its own mouth as opposed to those that it can understand through its incoming senses. When an object supports one or more outgoing interfaces, it is said to be *connectable.* In this chapter, we'll cover the mechanisms that make connectable objects—also, for brevity, called *sources*—work.

A source can, of course, have as many outgoing interfaces as it likes. Each interface is composed of distinct member functions, with each function representing a single *event, notification,* or *request.* Events and notifications are equivalent concepts (and interchangeable terms), as they are both used to tell the client that something interesting happened in the object—that data changed, a property changed, or the user did something such as click a button. Obviously events are very important for OLE Controls, which use the mechanisms we'll describe in this chapter. Events and notifications differ from a request in that the object expects no response from the client. A request, on the other hand, is how an object asks the client a question and expects a response. For example, an object that allows a client to override an action such as a property change will first ask the client whether it will allow